TenSQL: An SQL Database Built on GraphBLAS

Jon Roose
Autonomous Cyber Systems
Sandia National Laboratories
Albuquerque, New Mexico
jproose@sandia.gov

Miheer Vaidya
School of Computing
University of Utah
Salt Lake City, Utah
m.vaidya@utah.edu

Ponnuswamy Sadayappan
School of Computing
University of Utah
Salt Lake City, Utah
saday@cs.utah.edu

Sivasankaran Rajamanickam
Center for Computing Research
Sandia National Laboratories
Albuquerque, New Mexico
srajama@sandia.gov

***Abstract*—Relational Database Management Systems (RDBMS) have been the most prominent form of database in the world for several decades. While relational databases are often applied within high-frequency/low-volume transactional applications such as website backends, the poor performance of relational databases on low-frequency/high-volume queries often precludes their application to big data analysis fields like graph analytics. This work explores the construction of an RDBMS solution that uses the GraphBLAS API to execute Structured Query Language (SQL) in an effort to improve performance on high-volume queries. Tables are redefined to be collections of sparse scalars, vectors, matrices, and more generally sparse tensors. The explicit values (nonzeros) in these sparse tensors define the rows and NULL values within the tables. A prototype database called TenSQL was constructed and evaluated against several SQL implementations including PostgreSQL. Preliminary results comparing the performance on queries common in graph analysis applications offer performance improvements as high as 1,400x over PostgreSQL for moderately sized datasets when returning results in a columnar format.**

***Index Terms*—RDBMS, SQL, GraphBLAS, Graph Analytics, Sparse Linear Algebra, Databases, Query Optimization**

I. Introduction

Research and development (R&D) of data science systems, especially those involving graph analytics often leverage non-distributed systems. On the other hand, the data they analyze is often retrieved from distributed data storage solutions and cached locally to improve performance. In implementing these local caches, modern relational database management systems (RDBMS) such as PostgreSQL and SQLite are often eschewed in favor of less feature-rich but more performant storage options such as flat-files containing raw data dumps and Hierarchical Data Format version 5 (HDF5).

As data science tools transition towards production, moderately sized teams sometimes cobble together custom data storage solutions from several databases in order to accelerate particular portions of their data processing pipelines. For example, some teams leverage in-memory Redis databases to alleviate performance bottlenecks within RDBMS solutions. However, this practice makes it challenging to ensure atomicity and consistency among incoherent data stores.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

In an effort to improve the data consistency issues plaguing federated R&D environments that rely heavily upon cached data, we propose re-purposing the abstract sparse linear algebra concepts for graph algorithms pioneered by the GraphBLAS community [7], [10] to implement a new generation of RDBMS solutions optimized for the high-throughput/low-frequency workloads common to data science R&D. This work describes a proof-of-concept database called TenSQL that was constructed and benchmarked against several competitors to estimate potential performance improvements over existing industry standard RDBMS solutions. TenSQL was built upon the pygraphblas [14] module which internally leverages the SuiteSparse [5] reference implementation of the GraphBLAS C API [11]. While GraphBLAS has been used to express algorithms like pagerank, tringle counting, k-truss, and triangle centrality [4], [15], we demonstrate its utility in construction of RDBMS query engines as well.

The contributions of this paper are:

- Describing how to represent columnar RDBMS tables as collections of sparse tensors.
- Describing a method to execute a subset of SQL using the GraphBLAS API.
- Demonstrating that executing SQL queries using GraphBLAS operators can outperform industry standard databases by several orders of magnitude.

II. Related Work

First described in 2013, GraphBLAS describes a standardized set of linear algebra building blocks (kernels) that are useful for constructing graph analysis algorithms [10]. These kernels are exposed in libraries as highly optimized sparse linear algebra functions generalized to encompass alternative abstract algebra semirings. The flexibility to define semirings offered by GraphBLAS broadens its applicability to a much wider set of fields than traditional linear algebra packages. The first GraphBLAS C API specification was released in 2017 [11]. Since then, much effort has gone into building scalable and efficient GraphBLAS implementations for CPUs and GPUs [20].

Early in the development of GraphBLAS its progenitors discussed methods to use high performance implementations to build scalable user-friendly query engines for databases containing enormous datasets. For example, the Graphulo [6] project offers a set of sparse linear algebra operators for

the Apache Accumulo database. Additionally, Amossen [2], Kernert [8], and Qin [16] suggested borrowing optimizations from linear algebra to accelerate specific RDBMS queries. In contrast, TenSQL implements a new RDBMS solution from the ground up built upon sparse linear algebra and GraphBLAS.

Traditional RDBMS implementations such as PostgreSQL rely upon a relational model traditionally defined in terms of relation instances (tables) that are sets of tuples (rows) [17]. In contrast, TenSQL reasons about tables as a collection of columns, each represented by a sparse tensor. This attribute makes TenSQL a columnar relational database, similar to Amazon Redshift [1]. Whereas Redshift was built on top of PostgreSQL, TenSQL is built from scratch using Python and C libraries implementing the GraphBLAS API.

TileDB [12], [13] has a philosophically similar focus of storing datasets as tensors, although they prefer the term "array". TenSQL seeks to move beyond storage and retrieval of tensors by enabling native querying and processing of tensors using compute graphs built on linear algebra operators.

III. Mapping SQL Tables to Sparse Tensors

In this paper, we use the following notation: A tensor A of order $D_A \in \mathbb{Z}_{\geq 0}$ is described via three components: a shape vector $S_A \in \mathbb{N}^{D_A}$, an index set $X_A \subset (\mathbb{Z}_{\geq 0})^{D_A}$, and a value function $f_A : X_A \to \tau$. Traditional notation would describe such a tensor as $A \in \tau^{(S_A)_1 \times \ldots \times (S_A)_{D_A}}$.

In TenSQL, each table t is represented by several tensors of order D_t that have the same shape $S_t \in \mathbb{N}^{D_t}$. TenSQL sets each of the entries of this shape vector to 2^{60} by default, which is the largest size of tensor dimensions in SuiteSparse. Additionally, because the GraphBLAS API only handles scalars, vectors, and matrices, TenSQL requires $D_t \in \{0, 1, 2\}$.

Each row in every table is uniquely identified by a vector of integer primary key values of the form $x \in \left(0..(S_t)_1 - 1, ..., 0..(S_t)_{D_t} - 1\right)$ called the index of the row it identifies. Every entry in these index vectors corresponds to a column annotated with `PRIMARY KEY`. Therefore, D_t is equal to the number of primary key columns in t.

The collection of all row indices within a table is used to build a table's stencil χ_t with shape S_t and value function $f_{\chi_t}(x) := \texttt{true}$. The index of each nonzero element in χ_t uniquely identifies exactly one row in the table. If a table has no primary keys then it may contain at most one row identified by $\emptyset$.

For each column c within a table t that is not annotated with `PRIMARY KEY`, we instantiate one tensor V_c of shape S_t. The TenSQL prototype permits columns that are not primary keys to be integers, floats, or strings of various types. The type of column c determines the type of values stored in each explicit element (i.e. nonzero) of V_c. The set of indices corresponding to nonzero elements within V_c is required to be a subset of its table's stencil: $\texttt{indices}(V_c) \subseteq \texttt{indices}(\chi_t)$. An index in the stencil $x \in \texttt{indices}(\chi_t)$ that is absent from the column's tensor $x \notin \texttt{indices}(V_c)$ indicates a value of `NULL` in column c of the row identified by x. Therefore, if c is marked with `NOT NULL`, then we impose the requirement that $\texttt{indices}(V_c) = \texttt{indices}(\chi_t)$.

The TenSQL prototype implements string datatypes using the User Defined Type (UDT) feature of the GraphBLAS specification. Strings are allocated in Python, and references to their corresponding Python objects are stored as integer pointers within the table. When a string is added to a table the reference counter for the string is increased by one. An internal hash table mapping each Python objects' pointer to a 64-bit integer is used to track the number of times the same Python object is referenced within a table. Implementing strings this way enables TenSQL to provide unicode string functionality that is consistent with Python functionality.

The TenSQL prototype implements durability via functions that can save and load entire databases to and from HDF5 [18] files using the h5py Python module.

As a concrete example, the following table named `Dog` has four columns and four rows of data.

```
CREATE TABLE Dog (
        DogID INTEGER NOT NULL,
        Name TEXT,
        Age INTEGER,
        Weight REAL,
        PRIMARY KEY (DogID)
);
```

The data of the table `Dog` is provided in Figure 1. The age of two of the dogs (Bud and Rolf) is unknown, and so their age is represented as `NULL` in the database. The `Dog` table has one primary key column (`DogID`) and three columns that are not primary keys (`Name`, `Age`, `Weight`). Because `Dog` has 1 primary key column it is represented by sparse tensors of order 1 (vectors). As there are only four rows in the table, each vector may hold at most four explicit entries (i.e. nonzeros), although the actual vector sizes could be much larger to accommodate future insertions. The indices of the explicit entries of the special "stencil" tensor correspond to the values of the primary key column `DogID`. The remaining three columns (which are not primary keys) are each represented by a dedicated sparse vector. The empty box character indicates a sparse element whose value is not explicitly defined, which are often treated as implicit zeros by the linear algebra community. The vectors for the table `Dog` are provided in Figure 2.

DogID	Name	Age	Weight
0	"Spot"	4	31.1
1	"Bud"	NULL	77.5
2	"Shelby"	10	10.2
3	"Rolf"	NULL	80.0

Fig. 1: Rows of the `Dog` Table

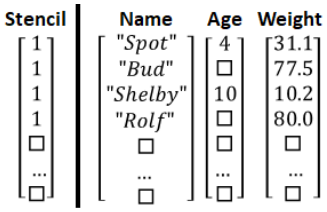

Fig. 2: Tensors of the `Dog` Table

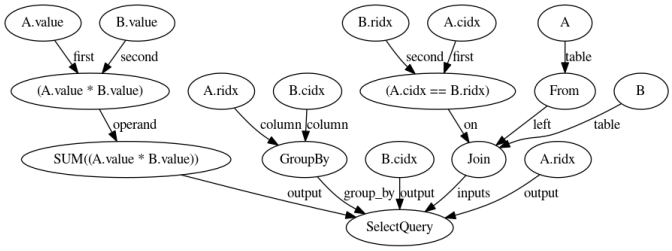

Fig. 4: An Example Query Intermediate Representation

IV. Query Execution

The overall data query language (DQL, a subset of SQL) execution process has the structure depicted in Figure 3.

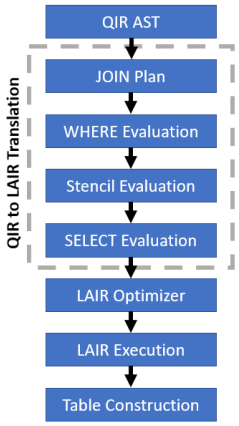

Fig. 3: TenSQL Data Query Language (DQL) Execution Flow

To aide understanding of how queries are executed, the following query which executes a sparse matrix multiply will be used.

```
SELECT A.ridx, B.cidx,
    SUM(A.value * B.value) AS value
FROM Matrix AS A
    JOIN Matrix AS B ON A.cidx = B.ridx
GROUP BY A.ridx, B.cidx;
```

A. Query Intermediate Representation

TenSQL does not implement a query parser. Instead, the user's Python script builds an abstract syntax tree (AST) in the format of an internal query intermediate representation (QIR). The QIR AST gets translated into internal Linar Algebra Intermediate Representation (LAIR) prior to execution. The QIR AST for the example query is shown in Figure 4.

B. JOIN Plan

TenSQL currently only implements `INNER JOIN`. It also requires that join conditions must only rely upon equality of primary key columns and `AND` conjunctions thereof. This restriction simplifies the process of developing join plans into identifying the connected components of an undirected graph where nodes are primary key columns and edges are equality constraints. Each connected component defines a primary key within the result table. The join plan for the exemplar query has three connected components. These are: (`A.ridx`), (`A.cidx`, `B.ridx`), and (`B.cidx`).

C. WHERE Evaluation

The QIR to LAIR translator uses the precomputed join plan to broadcast the tables referenced by the `WHERE` clause (if provided) onto a shared stencil. LAIR code is generated to evaluate the boolean expression described by the `WHERE` clause, and is then used to mask the shared stencil. Details of the broadcast operation are described in Section VI. The example query has no `WHERE` clause, so this masking is skipped.

D. Stencil Evaluation

If a `GROUP BY` clause is not provided, then the masked version of the shared stencil computed by the WHERE Evaluation phase is used as the output stencil. Otherwise, LAIR code is generated to reduce the shared stencil along axes whose associated primary key connected components are absent from the `GROUP BY` clause.

In the unoptimized LAIR of the example query (see Figure 5), the stencils of A and B are broadcast together by an InnerBroadcast operator. The `GROUP BY` clause results in a Reduction via logical OR over axis 2 of the stencil.

E. `SELECT` Expression Evaluation

Every `SELECT` clause must begin with one primary key column chosen from each of the join plan's connected components, except those omitted by the optional `GROUP BY` clause. If no `GROUP BY` clause is provided, the number of primary keys prefixing the `SELECT` statement must be equal to the number of connected components in the join plan.

For each of the remaining columns in the `SELECT` statement, LAIR code is generated to execute the requested arithmetic, boolean, and/or reduction operations using variants of the input columns that are broadcast to the shared stencil calculated by the Stencil Evaluation phase. Section VI describes the broadcast procedure. The LAIR translation is now complete.

In the unoptimized LAIR for the example query (see Figure 5), the broadcasting process introduces InnerBroadastMask operators that expand the matrices representing `A.value` and `B.value` into tensors of order 3. These sparse tensors are then multiplied in an elementwise fashion before summing over the final dimension using a reduction, resulting in sparse

matrices. Without further optimization, this would not be executable using GraphBLAS because of the intermediate result tensors of order 3.

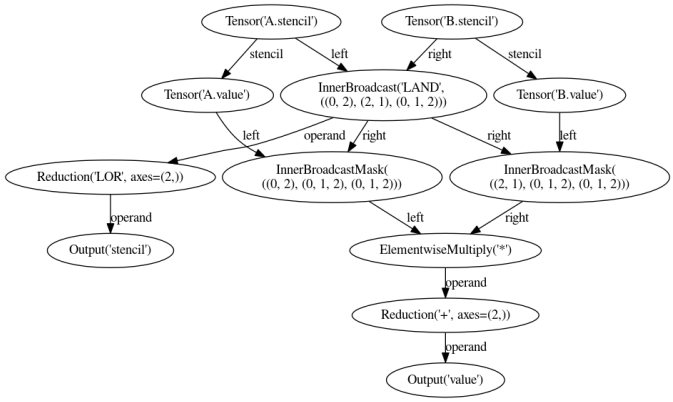

Fig. 5: Example LAIR (Unoptimized)

F. LAIR Optimizer

The generated LAIR AST is then optimized using two dataflow analysis passes and simple pattern matching algorithms. The first dataflow analysis pass, forward index set analysis, tracks intersections and unions of individual dimensions of tensor indices. Intersections arise from LAIR operators such as ElementwiseMultiply and unions arise from operators such as ElementwiseAdd. The second dataflow analysis pass, reverse index set analysis, uses the results of the forward index set analysis to trace back which indices will eventually be intersected and unioned with other indices. Taken together, in the event that the forward index set and reverse index set computed for InnerBroadcastMask, ElementwiseMultiply, and Mask nodes using certain monoids are equal, then those nodes are considered redundant and excised from the LAIR AST. This frequently allows the LAIR AST for `SELECT` expressions to be separated from the the LAIR AST generating output stencils, resulting in significant simplification.

Pattern matching is used for peephole optimizations, but is more importantly used to recognize applicability of GraphBLAS operators MxM and MxV within the LAIR AST. Special cases where `INNER JOIN` operations are combined with `GROUP BY` clauses and aggregation functions (e.g. `SUM` and `COUNT`) can result in InnerBroadcast operators followed by Reduction operators. In such cases, depending upon the number of primary keys in the tables of the columns referenced by the InnerBroadcast operator(s), this group of nodes can be replaced with a DotMxM, DotMxV, or DotVxM operator along with PermuteIndices operators for transpositions where necessary.

Optimizing the example query results in the LAIR AST visualized in Figure 6. The InnerBroadcastMask operations are eliminated because they are redundant with the ElementwiseMultiply that follows. Doing so decouples the stencil computation AST from the expression evaluation AST. However, the tensors must be promoted to order 3 according to the join plan, so the ElementwiseMultiply is simultaneously replaced with an InnerBroadcast operator. Finally, the pattern of an InnerBroadcast operator (with an appropriate broadcast pattern) followed by a Reduction operator is recognized, so two instances (one in the stencil AST and one in the expression evaluation AST) are replaced with DotMxM. This makes the final optimized AST executable, because all order-3 intermediate tensors are eliminated. This trick cannot be performed in every case, so queries resulting in order-3 tensors or higher must be detected and rejected by the TenSQL prototype.

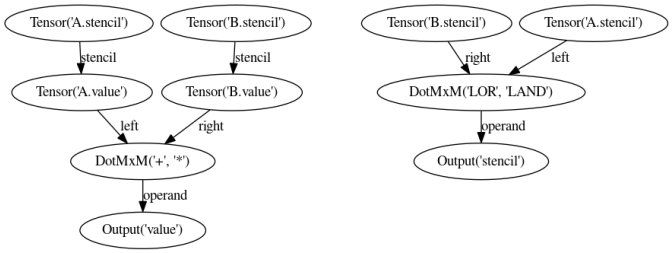

Fig. 6: Example LAIR (Optimized)

G. LAIR Execution

Once optimized, the LAIR AST is executed via a visitor class that invokes SuiteSparse kernels to compute result tensors.

H. Table Construction

The tensors generated by the `SELECT` clause expressions are then given names inferred from the QIR AST to produce column objects. These column objects are combined with the shared stencil to construct an unnamed Table object, which is returned as the query's result.

V. LAIR Operators

The allowable nodes in the current LAIR AST are:

DotMxM(left: Node, right: Node, add: **str**, mul: **str**)
Executes a matrix-matrix multiply. Generated by the LAIR optimizer.

DotMxV(left: Node, right: Node, add: **str**, mul: **str**)
Executes a matrix-vector multiply. Generated by the LAIR optimizer.

DotVxM(left: Node, right: Node, add: **str**, mul: **str**)
Executes a vector-matrix multiply. Generated by the LAIR optimizer.

ElementwiseAdd(op: **str**, left: Node, right: Node)
Given two operand tensors A, B returns tensor:

$$O : \left(S_A, X_A \cup X_B, f_O(x) := \begin{cases} f_{op}(f_A(x), f_B(x)) & x \in X_A \cap X_B \\ f_A(x) & x \in X_A \\ f_B(x) & x \in X_B \end{cases} \right)$$

ElementwiseApply(op: **str**, operand: Node)
Given an operand tensor A and an operator *op*, returns tensor:
$O : (S_A, X_A, f_O(x) := f_{op}(f_A(x)))$.

ElementwiseMultiply(op: **str**, left: Node, right: Node)
Given two operand tensors A, B, returns tensor:
$O : (S_A, X_A \cap X_B, f_O(x) := f_{op}(f_A(x), f_B(x)))$.

```
InnerBroadcast(op: str,
  left: Node, left_idx: Tuple[int],
  right: Node, right_idx: Tuple[int],
  out_idx: Tuple[int])
```
Broadcast tensors together using *op* according to the pattern (left_idx, right_idx, out_idx). See Section VI.

```
InnerBroadcastMask(
  left: Node, left_idx: Tuple[int],
  right: Node, right_idx: Tuple[int],
  out_idx: Tuple[int])
```
Mask one tensor with another according to the pattern (left_idx, right_idx, out_idx). See Section VI.

```
Mask(operand: Node, mask: Node)
```
Given an operand tensor A and mask tensor B, returns tensor: $O : (S_A, X_A \cap X_B, f_A)$.

```
Output(operand: Node, name: str)
```
Provides a name for the operand tensor that aids visualization of LAIR outputs.

```
Pattern(operand: Node)
```
Given an operand tensor A with boolean type, returns tensor: $O : (S_A, \{x : x \in X_A \cap f_A(x) \neq \texttt{false}\}, f_O(x) := \texttt{true})$.

```
PermuteIndices(operand: Node, out_axes: Tuple[int])
```
Permutes the index tuples of the sparse tensor operand according to the pattern specified in out_axes.

```
PrimaryKey(operand: Node, colnum: int, type_: Type)
```
Given an operand tensor A, generates a sparse tensor: $O : (S_A, X_A, f(x) = \texttt{type\_}(x_{\texttt{colnum}}))$

```
Reduction(op: str, operand: Node,
    reduce_axes: Tuple[int])
```
Reduces the operand tensor along the specified axes using operation *op*.

```
Tensor(operand: Tensor,
    stencil: Optional[Node] = None)
```
Imports a tensor into LAIR. The optional stencil argument asserts that $X_{\texttt{operand}} \subseteq X_{\texttt{stencil}}$.

VI. Broadcast Operations

TenSQL's LAIR describes broadcast operations that are essential for implementing `JOIN` clauses in SQL. Each broadcast operation combines a pair of tensors A, B into an output tensor O according to a broadcast pattern. Given a vector $h_O := [1, ..., D_O]^T$, broadcast patterns take the form $\left(P_A^{-1} W_A h_O, P_B^{-1} W_B h_O, h_O\right)$ where $P_A \in \mathbb{R}^{D_A \times D_A}, P_B \in \mathbb{R}^{D_B \times D_B}$ are permutation matrices, and $W_A \in \mathbb{R}^{D_A \times D_O}, W_B \in \mathbb{R}^{D_B \times D_O}$ are rectangular matrices formed by deleting rows from an identity matrix $\mathbf{I} \in \mathbb{R}^{D_O \times D_O}$. Each of these components is uniquely recoverable from the broadcast pattern.

For both InnerBroadcast and InnerBroadcastMask the shape and index set of the output tensor O are defined as:

$$S_O = W_A^T P_A S_A + W_B^T P_B S_B - W_{A \cap B}^T P_A S_A \tag{1}$$

$$X_O := \left\{ \begin{matrix} W_A^T P_A S_A + W_B^T P_B S_B - W_{A \cap B}^T P_A S_A : \\ x_A \in X_A, x_B \in X_B, W_{A \cap B}^T x_A = W_{B \cap A}^T x_B \end{matrix} \right\} \tag{2}$$

Where $W_{A \cap B} \in R^{D_A \times D_O}$ is formed from W_A, but with rows not in W_B replaced with zeros. Likewise, $W_{B \cap A} \in R^{D_B \times D_O}$ is formed from W_B, but with rows not in W_A replaced with zeros.

In addition to the operand tensors and broadcast pattern, InnerBroadcast accepts an operator $f_{op} : \texttt{range}(f_A) \times \texttt{range}(f_B) \to \texttt{range}(f_O)$. For implementation reasons inherited from GraphBLAS, InnerBroadcast requires that $\texttt{range}(f_A) = \texttt{range}(f_B) = \texttt{range}(f_O)$. The value function for the InnerBroadcast output tensor is:

$$f_O(x) := f_{op}\left(f_A(P_A^{-1} W_A x), f_B\left(P_B^{-1} W_B x\right)\right) \tag{3}$$

InnerBroadcastMask does not accept an operator and does not require that $\texttt{range}(f_A) = \texttt{range}(f_B)$. The value function for the InnerBroadcastMask output tensor is:

$$f_O(x) := f_A\left(P_A^{-1} W_A x\right) \tag{4}$$

The broadcast operators were implemented using a case statement composing appropriate GraphBLAS operators for every valid broadcast pattern where $D_A, D_B, D_O \in \{0, 1, 2\}$. This requires more than 20 specializations to implement.

VII. Limitations and Divergences

Although the TenSQL prototype implements a large subset of SQL, it omits some significant features. Descriptions of several of these limitations follow.

1) `OUTER JOIN` operations are not yet implemented because primary key columns are not allowed to contain `NULL` values. Creating matrices or vectors whose index sets contain `NULL` values is not well defined in traditional linear algebra. We plan to work around this issue by breaking column tensors into several tensors, each storing rows whose primary keys contain a particular combination of `NULL` values.
2) `ORDER BY` is not yet implemented because TenSQL's index sets are inherently unordered. We plan to add an optional ROWID sparse tensor to tables' metadata to track the ordering of tables' rows.
3) Join conditions are currently required to be composed of conjunctions of equality constraints on primary keys.
4) A SQL text parser, Data Description Language (DDL) queries such as `UPDATE` and `DELETE`, subqueries, transactions, foreign keys, unique constraints and indexing other than `PRIMARY KEY` are not yet implemented.
5) TenSQL natively returns results in a columnar format. Converting results to be row-wise requires an additional processing step.
6) All TenSQL operations currently take place in memory, with no support for out-of-core processing.

VIII. GraphBLAS Challenges

TenSQL also inherits some challenges from current GraphBLAS API design. Descriptions of some of these limitations and how TenSQL handles these follow.

1) The GraphBLAS API cannot manipulate sparse tensors of order greater than two. Therefore TenSQL's tables, result sets, and intermediate results are limited to at most two primary keys so that $D_t \in \{0, 1, 2\}$ for all tensors.

Provided a sufficient level of expertise, this restriction can often be worked around when crafting SQL schemas and queries, but it makes some use cases difficult or even impossible to implement. Sparse tensor extensions to the GraphBLAS API would resolve this challenge.

2) The GraphBLAS API has no inner broadcast operator. TenSQL works around this limitation through composition of other operators such as mxm and apply_bind_first. However, doing so requires a series of if statements with more than 20 branches to account for various tensor orders and transpositions, which will not scale to higher order tensors.
3) The GraphBLAS API does not offer string datatypes for tensor elements. TenSQL works around this limitation by casting python objects containing strings to integer pointers and internally counting references to these objects for garbage collection purposes.
4) The GraphBLAS API does not provide for certain operations on scalars (e.g. apply, eWiseAdd, and eWiseMult). To work around this limitation, scalars frequently need to be handled in dedicated code branches. Compounding the problem, the GraphBLAS API does not provide a mechanism to call operators directly, so TenSQL re-implements these existing operators in Python code. This leads to potential inconsistences in how things like rounding, floating-point exceptions, and integer overflow are handled depending upon the order of the tensor.

IX. Benchmarks

Three benchmarks were implemented within TenSQL to compare its performance on graph-relevant tasks against industry standard competitors such as PostgreSQL and SQLite.

A. Datasets and Schema

To provide a sense of scaling, all three benchmarks were executed against three ego-network datasets from the Stanford Network Analysis Project (SNAP) [9] based on data taken from several social media websites. The smallest is the Facebook dataset, which is a graph with 4,096 nodes and 88,234 undirected edges. The medium sized dataset comes from Twitter, with 81,306 nodes and 1,768,149 directed edges. Finally, the Google+ dataset contains 107,614 nodes and 13,673,453 directed edges. Adjacency matrices were constructed from these graph data structures to provide a dataset for each benchmark. Random numbers between 0 and 1 were used to generate edge weights.

For PostgreSQL, and SQLite the following tables were created to support benchmarking:

```
CREATE TABLE Node (
        idnode NOT NULL,
        guid VARCHAR(36) NOT NULL,
        PRIMARY KEY (idnode)
);

CREATE TABLE Edge (
        first INTEGER NOT NULL,
        second INTEGER NOT NULL,
        value FLOAT NOT NULL,
        PRIMARY KEY (first, second)
);
```

The schema for TenSQL is almost identical, except that it uses BigInt column types for the primary key columns. Adding indices to the `Edge` table's first and second columns was attempted, but did not improve performance on any of these benchmarks.

B. Implementation Details

The guiding principle in these benchmarks' design was to simulate common graph analytics use cases. In all cases, the benchmarks were executed on a single node running an AMD EPYC 7543P 32-Core processor with 512GB of memory. The node was also equipped with Kioxia CD6 960GB TLC NVMe SSD's connected via PCIe 4.0. The SQLAlchemy middleware package was used to keep implementation as simple and consistent as possible. PostgreSQL 15.2 was used in the benchmarks. The configuration was based on PGTune [19] recommendations. TenSQL used version 5.1.8.0 of the pygraphblas library, and version 7.3.3 of SuiteSparse.

C. Durable Ingest Benchmark

The ingest benchmark reveals the runtime required to ingest the Edge and Node tables, using randomized (but precomputed) UUID's for the `Node.guid` column. This represents a common first step in real-world use cases when working with graph datasets. The following parameterized queries were generated by SQLAlchemy to implement the benchmark:

```
INSERT INTO Edge (first, second, value)
VALUES (%(first)s, %(second)s, %(value)s);

INSERT INTO Node (idnode, guid)
VALUES (%(idnode)s, %(guid)s);
```

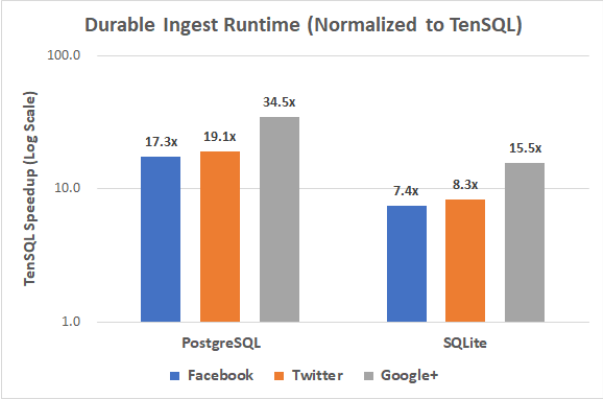

Fig. 7: TenSQL Durable Ingest Speedup

For the largest dataset (SNAP Goolge+) TenSQL ingested the dataset within 6.37 seconds, while PostgreSQL required 3 minutes and 40 seconds. SQLite was faster than PostgreSQL, requiring 1 minute 39 seconds. Thus, TenSQL was 34.5x faster than PostgreSQL, and 15.5x faster than SQLite on this benchmark.

D. Two-Hop Benchmark

The sparse matrix multiply (MxM) kernel is used in subgraph extraction technqiues such as breadth-first search (BFS) [7]. This benchmark is designed to use an MxM to extract the two-hop neighbors of all nodes. The following query was used to implement the benchmark:

```
SELECT A.first , B.second
FROM Edge AS A
    JOIN Edge AS B ON A.second = B.first
GROUP BY A.first , B.second;
```

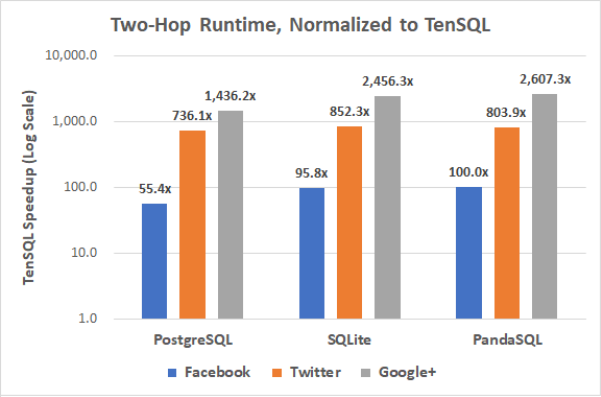

Fig. 8: TenSQL Two-Hop Speedup

For the largest dataset (SNAP Google+) TenSQL required 2.14 seconds to execute this query, while PostgreSQL required 51 minutes and 9 seconds. Thus, TenSQL outperformed PostgreSQL by a factor of 1,436x. TenSQL's results are returned in a columnar fashion. Where row-wise results are desired an additional post-processing step must be applied.

E. Named Edges Benchmark

The named edges benchmark queries the edge list of a graph with added context from the graph's nodes. This reflects a common real-world use case where external identifiers of nodes (and other per-node information) are stored in a table that is separate from the edge table. This is often done as part of the schema normalization process to reduce the size of databases containing graphs by minimizing data redundancy.

```
SELECT
    x.guid AS first , y.guid AS second
FROM Edge AS "A"
    JOIN Node AS x ON "A".first = x.idnode
    JOIN Node AS y ON "A".second = y.idnode;
```

For the largest dataset (SNAP Google+), columnar TenSQL required 39.2 seconds to plan and execute this query, while PostgreSQL required 32.5 seconds. Therefore, PostgreSQL was 1.21x faster than TenSQL on this query. However, when row-wise results are required, TenSQL takes 1 minute 14 seconds to execute the query. In such a case, PostgreSQL's runtime advantage over TenSQL widens to 2.29x.

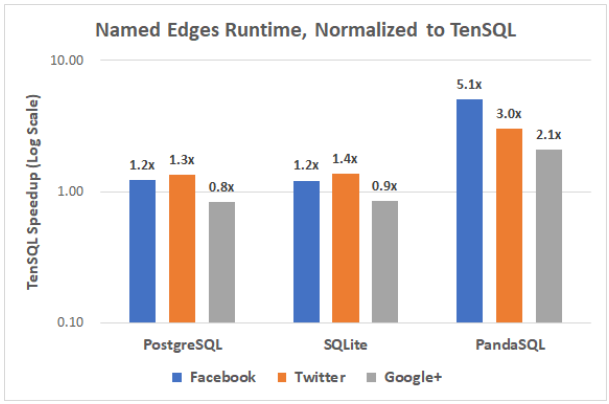

Fig. 9: TenSQL Named Edges Speedup

X. Future Work

Follow-on efforts are likely to include building and benchmarking tools that translate additional query languages to LAIR. Such efforts may lead to a multi-lingual database capability that can execute relational queries via SQL or dataframes, graph queries via SPARQL Protocol and RDF Query Language (SPARQL) or Neo4j's Cypher Query Language (CQL), and linear algebra operations via MATLAB syntax or similar. CQL, in particular, was recently implemented in the RedisGraph extension to the Redis database using GraphBLAS operators [3]. Experiments with using GPUs to accelerate queries translated to LAIR may also follow, given SuiteSparse's support for GPUs.

XI. Conclusion

This paper describes the design, construction, and prototyping of Structured Query Language (SQL) operations on top of GraphBLAS API compliant sparse linear algebra libraries. Although many features of SQL are not implemented by the current prototype, it offers a framework for compiling practically useful portions of SQL into sparse linear algebra building blocks. When implemented using the SuiteSparse reference implementation of GraphBLAS and returning columnar results, the prototype executes certain queries as much as 1,400x faster than PostgreSQL on relatively modest graph datasets. Further work in this direction may make it feasible to rapidly implement practically useful subsets of SQL on additional hardware and software systems using GraphBLAS.

XII. Acknowledgements

We would like to express gratitude to those who provided constructive feedback on project direction, notation, and editing. Those who provided feedback include: Jon Berry, Michael Eydenberg, Casey Haynes, Atanas Rountev, and Keshav Sreekumar.

References

[1] Amazon.com, Inc. What is a columnar database?, 2023.

[2] Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126, 2009.

[3] Pieter Cailliau, Tim Davis, Vijay Gadepally, Jeremy Kepner, Roi Lipman, Jeffrey Lovitz, and Keren Ouaknine. RedisGraph GraphBLAS Enabled Graph Database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 285–286, 2019.

[4] Timothy A Davis. Graph algorithms via SuiteSparse: GraphBLAS: triangle counting and k-truss. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, page 3. IEEE, 2018.

[5] Timothy A. Davis. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *ACM Trans. Math. Softw.*, 45(4), dec 2019.

[6] Vijay Gadepally, Jake Bolewski, Dan Hook, Dylan Hutchison, Ben Miller, and Jeremy Kepner. Graphulo: Linear algebra graph kernels for nosql databases. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 822–830. IEEE, 2015.

[7] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[8] David Kernert, Frank Köhler, and Wolfgang Lehner. SLACID-sparse linear algebra in a column-oriented in-memory database system. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2014.

[9] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection, June 2014.

[10] Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, Jeremy Kepner, Charles Leiserson, Andrew Lumsdaine, David Padua, Stephen Poole, Steve Reinhardt, Mike Stonebraker, Steve Wallach, and Andrew Yoo. Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2, 2013.

[11] Timothy G Mattson, Carl Yang, Scott McMillan, Aydin Buluç, and José E Moreira. GraphBLAS C API: Ideas for future versions of the specification. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2017.

[12] Stavros Papadopoulos. A deep dive into the TileDB data format & storage engine, 2023.

[13] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The TileDB array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, 2016.

[14] Michel Pelletier. pygraphblas: GraphBLAS for python, 2019-2021.

[15] Michel Pelletier, Will Kimmerer, Timothy A Davis, and Timothy G Mattson. The GraphBLAS in Julia and Python: the PageRank and Triangle Centralities. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2021.

[16] Chengjie Qin and Florin Rusu. Dot-product join: Scalable in-database linear algebra for big model analytics. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2017.

[17] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. Computer science series. McGraw-Hill, second edition, 2000.

[18] The HDF Group. Hierarchical data format version 5, 1997-2023.

[19] Oleksii Vasyliev. PGTune, 2022.

[20] Carl Yang, Aydın Buluç, and John D Owens. GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. *ACM Transactions on Mathematical Software (TOMS)*, 48(1):1–51, 2022.